Estratégia Concursos

Aula 01

IFCE (Prof. Ciência da Computação-Metodologia e Técnicas da Comput) Conhecimentos - 2021(Pós-Edital)

Autor:

Diego Carvalho, Equipe Informática e TI, Evandro Dalla Vecchia Pereira , Pedro Henrique Chagas Freitas, Thiago Rodrigues

Cavalcanti, Raphael Henrique Lacerda

Aula 01

20 de Setembro de 2021

Sumário

1 COMPLEXIDADE DE ALGORITMOS

1.1 INTRODUÇÃO

Galera, por que estudamos a complexidade de algoritmos? Para determinar o custo computacional (tempo, espaço, etc) para execução de algoritmos. Em outras palavras**, ela classifica problemas computacionais de acordo com sua dificuldade inerente**. É importante entender isso para posteriormente estudarmos a complexidade de métodos de ordenação e métodos de pesquisa.

Nosso estudo aqui será bastante superficial por duas razões: primeiro, concursos cobram pouco e, quando cobram, querem saber as complexidades dos métodos de ordenação mais conhecidos; segundo, essa é uma disciplina absurdamente complexa que envolve Análise Assintótica, Cálculo Diferencial, Análise Polinomial (Linear, Exponencial, etc). **Logo, vamos nos ater ao que cai em concurso público**!

Só uma pausa: passei dias sem dormir na minha graduação por conta dessa disciplina! Na UnB, ela era ministrada pelo Instituto de Matemática e era considerada a disciplina mais difícil do curso :-(. Continuando: lá em cima eu falei sobre custo computacional! **Ora, para que eu escolha um algoritmo, eu preciso definir algum parâmetro**.

Podemos começar com Tempo! *Um algoritmo que realiza uma tarefa em 20 minutos é melhor do que um algoritmo que realiza uma tarefa em 20 dias?* **Não é uma boa estratégia, porque depende do computador que eu estou utilizando** (e todo o hardware correspondente), depende das otimizações realizadas pelo compilador, entre outras variáveis.

Vamos analisar, então, Espaço! *Um algoritmo que utiliza 20Mb de RAM é melhor do que um algoritmo que utiliza 20Gb?* Seguem os mesmos argumentos utilizados para o Tempo, ou seja, não é uma boa opção! *E agora, o que faremos?* Galera, eu tenho uma sugestão: **investigar a quantidade de vezes que operações são executadas na execução do algoritmo**!

Essa estratégia independe do computador (e hardware associado), do compilador, da linguagem de programação, das condições de implementação, entre outros fatores – ela depende apenas da qualidade inerente do algoritmo[1] implementado. **Utilizam-se algumas simplificações matemáticas para se ter uma ideia do comportamento do algoritmo**. Prosseguindo...

Dada uma entrada de dados de tamanho N, podemos calcular o custo computacional de um algoritmo em seu pior caso, médio caso e melhor caso! *Como assim, professor?* Para entender isso, vamos utilizar a metáfora de um jogo de baralho! Imaginem que eu estou jogando contra vocês. Vocês embaralham e me entregam 5 cartas, eu embaralho novamente e lhes entrego 5 cartas.

Quem joga baralho sabe que uma boa alternativa para grande parte dos jogos é ordenar as cartas em ordem crescente de modo a encontrar mais facilmente a melhor carta para jogar. Agora observem... vocês receberam as seguintes cartas (nessa ordem): 4, 5, 6, 7, 8. Já eu recebi as seguintes cartas

[1] Pessoal, é claro que nossa visão sobre a complexidade dos algoritmos é teórica. Na prática, depende de diversos outros fatores, mas nosso foco é na visão analítica e, não, empírica.

(também nessa ordem): 8, 7, 6, 5, 4 – **nós queremos analisar a complexidade de ordenação dessas cartas**.

Ora, convenhamos que vocês possuem o melhor caso, porque vocês deram a sorte de as cartas recebidas já estarem ordenadas. Já eu peguei o pior caso, porque as cartas estão ordenadas na ordem inversa. Por fim, o caso médio ocorre caso as cartas recebidas estejam em uma ordem aleatória. Com isso, espero que vocês tenham entendido o sentido de pior, médio e melhor casos.

Vamos partir agora para o estudo da Notação Big-O (ou Notação Assintótica)! Isso é simplesmente uma forma de representar o comportamento assintótico de uma função. No nosso contexto, ela busca expressar a quantidade de operações primitivas executadas como função do tamanho da entrada de dados. Vamos ver isso melhor!

A Notação Big-O é a representação relativa da complexidade de um algoritmo. É relativa porque só se pode comparar maçãs com maçãs, isto é, você não pode comparar um algoritmo de multiplicação aritmética com um algoritmo de ordenação de inteiros. **É uma representação porque reduz a comparação entre algoritmos a uma simples variável por meio de observações e suposições**.

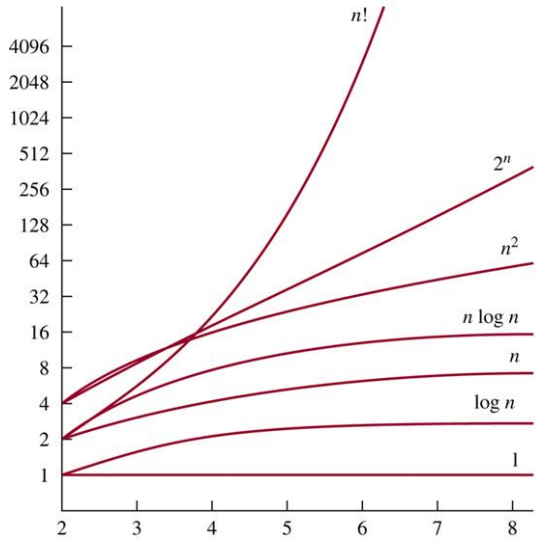
E trata da complexidade porque se é necessário 1 segundo para ordenar 10.000 elementos, quanto tempo levará para ordenar 1.000.000? **A complexidade, nesse exemplo particular, é a medida relativa para alguma coisa.** Vamos ver isso por meio de um exemplo: soma de dois inteiros! A soma é uma operação ou um problema, e o método para resolver esse problema é chamado algoritmo!

Transporte →	1	1	1	1	0
Parcela 1 →		1	1	0	1
Parcela 2 →	+	1	0	1	1
Soma →	1	1	0	0	0

Vamos supor que no algoritmo de somar (mostrado acima), a operação mais cara seja a adição. Observem que se somarmos dois números de 100 dígitos, teremos que fazer 100 adições. Se somarmos dois números de 10.000 dígitos, teremos que fazer 10.000 adições. *Perceberam o padrão?* **A complexidade (aqui, número de operações) é diretamente proporcional ao número *n* de dígitos, i.e., O(n)**.

Quando dizemos que um algoritmo é $O(n^2)$, estamos querendo dizer que esse algoritmo é da ordem de grandeza quadrática! Ele basicamente serve para te dizer quão rápido uma função cresce, por exemplo: um algoritmo $O(n)$ é melhor do que um algoritmo $O(n^2)$, porque ela cresce mais lentamente! **Abaixo vemos uma lista das classes de funções mais comuns em ordem crescente de crescimento**:

Notação	Nome
$O(1)$	Constante
$O(\log n)$	Logarítmica
$O[(\log n)^c]$	Polilogarítmica
$O(n)$	Linear
$O(n \log n)$	–
$O(n^2)$	Quadrática
$O(n^3)$	Cúbica
$O(n^c)$	Polinomial
$O(c^n)$	Exponencial
$O(n!)$	Fatorial



Quando dizemos que o Shellsort é um algoritmo $O(n^2)$, estamos querendo dizer que a complexidade (nesse caso, o número de operações) para ordenar um conjunto de n dados com o Algoritmo Shellsort é proporcional ao quadrado do número de elementos no conjunto! **Grosso modo, para ordenar 20 números, é necessário realizar 400 operações** (sem entrar em detalhes sobre a operação em si, nem sobre as simplificações matemáticas que são realizadas).

Entender como se chega a esses valores para cada método de ordenação e pesquisa é extremamente complexo! Galera, apesar de eu nunca ter visto isso em prova, é bom que vocês saibam que existem outras notações! Utiliza-se Notação Big-O (O) para pior caso; Notação Big-Ômega para melhor caso (Ω); e Notação Big-Theta (Θ) para caso médio.

Como na prática utiliza-se Big-O para tudo, o que eu recomendo (infelizmente, porque eu sei que vocês têm zilhões de coisas para decorar) é memorizar o pior caso dos principais métodos. Dessa forma, é possível responder a maioria das questões de prova sobre esse tema. Eventualmente, as questões pedem também caso médio e melhor caso, mas é menos comum. *Bacana?* :-)

Por último, uma pergunta muito frequente: Professor, já vi questões cobrando Logaritmo na Base 10, Logaritmo na Base 2, Logaritmo Neperiano, etc... *isso não está errado?* Galera, suponha que um algoritmo levou um tempo $\Theta(\log_{10} n)$. Você também poderia dizer que levou um tempo $\Theta(\lg n)$ (ou seja, $\Theta(\log_2 n)$). **Você pode utilizar qualquer base.**

Sempre que a base do logaritmo é uma constante, não importa a base que usamos na notação assintótica. *Por que não?* Porque, para a notação assintótica, isso é completamente irrelevante. Beleza? Então, não se prendam a base do logaritmo, qualquer uma pode ser utilizada na representação de complexidade assintótica de algoritmos. Bacana? Exercícios...

1.2 EXERCÍCIOS COMENTADOS: COMPLEXIDADE DE ALGORITMOS

1. (FGV / Analista Censitário (IBGE) / 2017 / Desenvolvimento de Aplicações - WEB Mobile / Análise de Sistemas)

Para projetar algoritmos eficientes um desenvolvedor deve estar preocupado com a complexidade deste algoritmo, desde sua concepção.

Considere a seguinte função T(n) que mede os recursos (ex. tempo de execução) que um algoritmo necessita no pior caso para processar uma entrada qualquer de tamanho n:

T(n) = O(log(n))

Sabendo que O(log(n)) é a ordem da complexidade de tempo do algoritmo seguindo a notação "big O", é correto afirmar que este algoritmo tem complexidade de ordem:
a) constante;
b) sublinear;
c) linear;
d) polinomial;
e) exponencial.

Gabarito: **Letra B**.

a) constante;
ERRADO - A complexidade é constante se T(n) = **O(1)**.

b) sublinear;
CERTO - O termo sublinear está sendo utilizado aqui com "menor que a linear". A função y = log(n) tem crescimento menor que y = n. O nome exato dessa complexidade é **Logarítmica**, mas não deixa de ser uma complexidade abaixo da linear, portanto, sublinear.

c) linear;
ERRADO - A complexidade é linear se T(n) = **O(n)**.

d) polinomial;
ERRADO - A complexidade é polinomial se T(n) = **$O(n^c)$**.

e) exponencial.
ERRADO - A complexidade é exponencial se T(n) = **$O(c^n)$**.

2. (FCC / Analista Judiciário (TRF 5ª Região) / 2017 / Informática - Desenvolvimento / Apoio Especializado)

Considere o algoritmo abaixo.

```
static int fibonacci(int n) {
  if (n <= 1) {
     return n;
  }
  return fibonacci(n - 2) + fibonacci(n - 1);
}
```

A complexidade deste algoritmo, na notação Big O, é
a) $O(2^n)$.
b) $O(n^2)$.
c) $O(n)$.
d) $O(\log(n))$.
e) $O(n^4)$.

Gabarito: **Letra A**.

A chamada da função fibonacci gera, potencialmente, 2 chamadas para a mesma função. Essas 2 chamadas geram, cada uma 2 chamadas, tornando-se, potencialmente, 4. Essas 4 geram 2 cada uma, tornando-se potencialmente 8. A cada iteração, dobramos.

Logo, após n iterações, faremos 2^n chamadas, o que significa que a complexidade é $O(2^n)$, **Letra A**.

3. (FGV / Analista Legislativo (ALERO) / 2018 / Infraestrutura de Redes e Comunicação / Tecnologia da Informação)

Considere a Sequência de Fibonacci (0, 1, 1, 2, 3, 5, 8, 13, ...), onde os dois primeiros termos valem 0 e 1 respectivamente, e cada termo seguinte é a soma de seus dois predecessores.

O pseudocódigo a seguir apresenta um algoritmo simples para o cálculo do N-ésimo termo dessa sequência.

```
function fibo (N)
if n = 1 then
     return 0
elif n = 2 then
     return 1
else
       penultimo := 0
       ultimo := 1
       for i := 3 until N do
            atual := penultimo + ultimo
            penultimo := ultimo
            ultimo := atual
       end for
       return atual
```

```
end if
```

Assinale a opção que mostra a complexidade desse algoritmo.
a) O(n/2)
b) O(n)
c) O(n2)
d) O(log n)
e) O(2n)

Gabarito: **Letra B**.

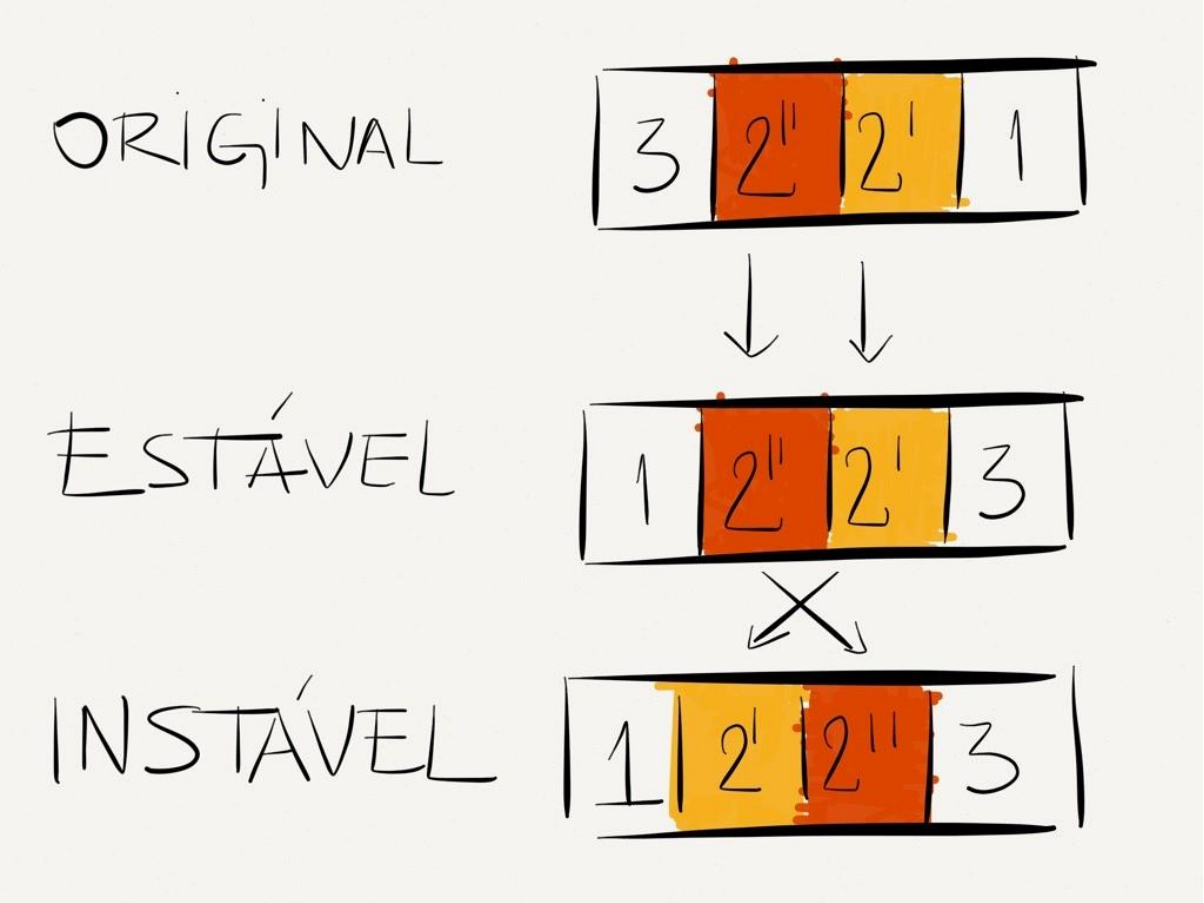
O algoritmo não usa recursividade, e possui apenas um loop. Esse loop itera de 3 até N. Portanto, o número de operações será proporcional a N, gerando uma complexidade O(N) - **Letra B**.

2 MÉTODOS DE ORDENAÇÃO

2.1 INTRODUÇÃO

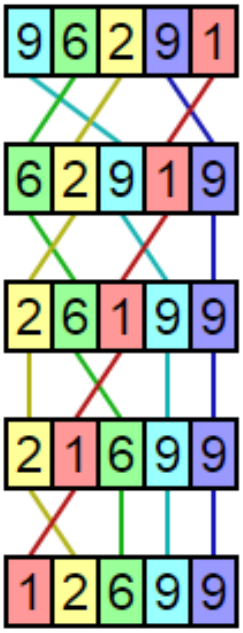
Métodos de Ordenação são algoritmos que têm o objetivo principal de posicionar os elementos de uma estrutura de dados em uma determinada ordem. *Para que, professor?* Ora, isso possibilita o acesso mais rápido e eficiente aos dados. Existem dezenas de métodos, todavia nessa aula veremos apenas os mais importantes: BubbleSort, QuickSort, InsertionSort, SelectionSort, MergeSort, ShellSort e HeapSort.

Antes de iniciar, vamos falar sobre o conceito de Estabilidade! Um método estável é aquele em que os itens com chaves iguais se mantêm com a posição inalterada durante o processo de ordenação, i.e., preserva-se a ordem relativa dos itens com chaves duplicadas ou iguais. **Métodos Estáveis: Bubble, Insertion e Merge; Métodos Instáveis: Selection, Quick, Heap e Shell**. Vejamos um exemplo:



Na imagem acima, foi colocado um sinal de aspas simples e duplas apenas para diferenciá-los, mas trata-se do mesmo número. Um algoritmo estável ordena todo o restante e não perde tempo trocando as posições de elementos que possuam chaves idênticas. Já um algoritmo instável ordena todos os elementos, inclusive aqueles que possuem chaves idênticas (sob algum outro critério).

2.2 BUBBLE SORT (2 A 2)



Esse algoritmo é o primeiro método de ordenação aprendido na faculdade, porque ele é bastante simples e intuitivo. Nesse método, os elementos da lista são movidos para as posições adequadas de forma contínua. Se um elemento está inicialmente em uma posição *i* e, para que a lista fique ordenada, ele deve ocupar a posição *j*, então ele terá que passar por todas as posições entre *i* e *j*.

Em cada iteração do método, percorremos a lista a partir de seu início comparando cada elemento com seu sucessor, trocando-se de posição se houver necessidade. É possível mostrar que, se a lista tiver *n* elementos, após no máximo *(n-1)* iterações, a lista estará em ordem (crescente ou decrescente). Observem abaixo o código para a Ordenação em Bolha:

```
ALGORITMO BOLHA
  ENTRADA: UM VETOR L COM N POSIÇÕES
  SAÍDA: O VETOR L EM ORDEM CRESCENTE

  PARA i = 1 até n - 1
    PARA j = 0 até n - 1 - i
      SE  L[j] >  L[j+1]
        AUX = L[j]          // SWAP
        L[j] = L[j+1]
        L[j+1] = AUX
FIM {BOLHA}
```

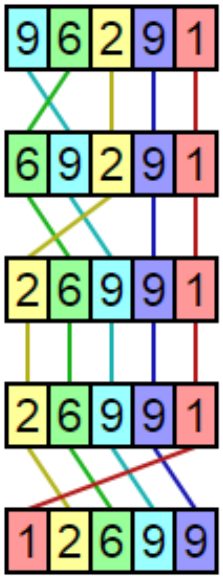
Melhor Caso	Caso Médio	Pior Caso
$\Omega(n)$	$\Omega(n^2)$	$\Omega(n^2)$

2.3 INSERTION SORT (INSIRA)

Esse algoritmo, também conhecido como Inserção Direta, é bastante simples **e apresenta um desempenho significativamente melhor que o BubbleSort, em termos absolutos**. Além disso, ele é extremamente eficiente para listas que já estejam substancialmente ordenadas e listas com pequeno número de elementos. Observem abaixo o código para a Ordenação de Inserção:

```
ALGORITMO INSERÇÃO
  ENTRADA: UM VETOR L COM N POSIÇÕES
  SAÍDA: O VETOR L EM ORDEM CRESCENTE

  PARA i = 1 até n - 1
    PIVO = L[i]
    j = i - 1
    ENQUANTO j ≥ 0 e L[j] > PIVO
      L[j+1] = L[j]
      j = j - 1
    L[j+1] = PIVO
FIM {INSERÇÃO}
```

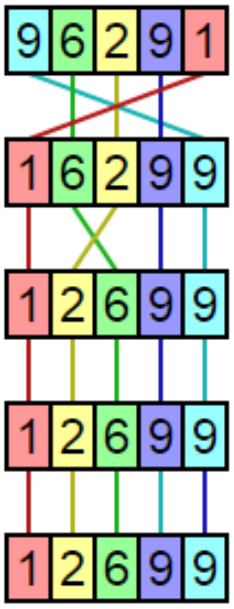


Nesse método, consideramos que a lista está dividida em parte esquerda, já ordenada, e parte direita, em possível desordem. Inicialmente, a parte esquerda contém apenas o primeiro elemento da lista. **Cada iteração consiste em inserir o primeiro elemento da parte direta (pivô) na posição adequada da parte esquerda, de modo que a parte esquerda continue ordenada**.

É fácil perceber que se a lista possui *n* elementos, após (n-1) inserções, ela estará ordenada. Para inserir o pivô, percorremos a parte esquerda, da direita para a esquerda, deslocando os elementos estritamente maiores que o pivô uma posição para direita. **O pivô deve ser colocado imediatamente à esquerda do último elemento movido**.

Melhor Caso	Caso Médio	Pior Caso
$\Omega(n)$	$\Omega(n^2)$	$\Omega(n^2)$

2.4 SELECTION SORT (SELECIONE)



Esse algoritmo consiste em selecionar o menor[2] elemento de um vetor e trocá-lo *(swap)* pelo item que estiver na primeira posição, i.e., inseri-lo no início do vetor. Essas duas operações são repetidas com os itens restantes até o último elemento. Tem como ponto forte o fato de que realiza poucas operações de *swap*. Seu desempenho costuma ser superior ao BubbleSort e inferior ao InsertionSort.

Assim como no InsertionSort, considera-se que a lista está dividida em parte esquerda, já ordenada, e parte direita, em possível desordem. Ademais, os elementos da parte esquerda são todos menores ou iguais aos elementos da parte direita. **Cada iteração consiste em selecionar o menor elemento da parte direita (pivô) e trocá-lo com o primeiro elemento da parte direita.**

Assim, a parte esquerda aumenta, visto que passa a incluir o pivô, e a parte direita diminui. Note que o pivô é maior que todos os demais elementos da parte esquerda, portanto a parte esquerda continua ordenada. Ademais, o pivô era o menor elemento da direita, logo todos os elementos da esquerda continuam sendo menores ou iguais aos elementos da direita. Inicialmente, a parte esquerda é vazia.

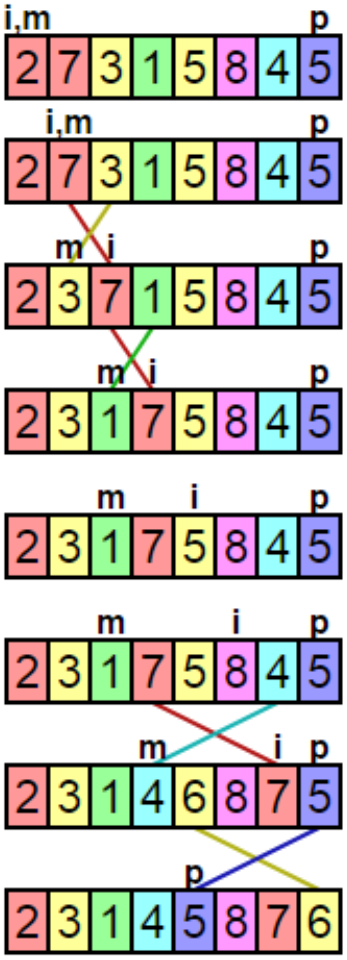
```
ALGORITMO SELEÇÃO
  ENTRADA: UM VETOR L COM N POSIÇÕES
  SAÍDA: O VETOR L EM ORDEM CRESCENTE

  PARA i = 0 ate n - 2
    MIN = i
    PARA j = i + 1 até n - 1
      SE L[j] < L[MIN]
        MIN= j
    TROCA(L[i], L[MIN])
FIM {SELEÇÃO}
```

Melhor Caso	Caso Médio	Pior Caso
$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$

[2] A definição formal afirma que é o maior valor; a maioria das implementações utiliza o menor valor. As questões de prova cobram algumas vezes o maior, outras vezes o menor.

2.5 QUICKSORT (PIVÔ)



Esse algoritmo divide um conjunto de itens em conjuntos menores, que são ordenados de forma independente, e depois os resultados são combinados para produzir a solução de ordenação do conjunto maior. **Trata-se, portanto, de um algoritmo do tipo *Divisão-e-Conquista*, i.e., repartindo os dados em subgrupos, dependendo de um elemento chamado pivô**.

Talvez seja o método de ordenação mais utilizado! Isso ocorre porque quase sempre ele é significativamente mais rápido do que todos os demais métodos de ordenação baseados em comparação. **Ademais, suas características fazem com que ele, assim como o MergeSort, possa ser facilmente paralelizado.** Ele também pode ser adaptado para realizar ordenação externa (QuickSort Externo).

Neste método, a lista é dividida em parte esquerda e parte direita, sendo que os elementos da parte esquerda são todos menores que os elementos da parte direita. Essa fase do processo é chamada de partição. **Em seguida, as duas partes são ordenadas recursivamente (usando o próprio QuickSort). A lista está, portanto, ordenada corretamente**!

Uma estratégia para fazer a partição é escolher um valor como pivô e então colocar na parte esquerda os elementos menores ou iguais ao pivô e na parte direita os elementos maiores que o pivô – galera, a escolha do pivô é crítica! **Em geral, utiliza-se como pivô o primeiro elemento da lista, a despeito de existirem maneiras de escolher um "melhor" pivô**.

Esse algoritmo é um dos métodos mais rápidos de ordenação, apesar de às vezes partições desequilibradas poderem conduzir a uma ordenação lenta. A eficácia do método depende da escolha do pivô mais adequado ao conjunto de dados que se deseja ordenar. Alguns, por exemplo, utilizam a mediana de três elementos para otimizar o algoritmo.

Alguns autores consideram a divisão em três subconjuntos, sendo o terceiro contendo valores iguais ao pivô. O Melhor Caso ocorre quando o conjunto é dividido em subconjuntos de mesmo tamanho; o Pior Caso ocorre quando o pivô corresponde a um dos extremos (menor ou maior valor). Alguns o consideram um algoritmo frágil e não-estável, com baixa tolerância a erros.

```
PROCEDIMENTO PARTIÇÃO
  ENTRADA: UM VETOR L E AS POSIÇÕES INICIO E FIM
  SAÍDA:j, tal que L[INICIO]..L[j-1] ≤ L[j] e
        L[j+1]..L[FIM] > L[j]
     // j é a posição onde será colocado o pivô, como
     // ilustrado na figura abaixo
       PIVO = L[INICIO], i = INICIO + 1, j = FIM
        ENQUANTO i ≤ j
          ENQUANTO i ≤ j e L[i] ≤ PIVO
            i = i + 1
          ENQUANTO L[j] > PIVO
            j = j - 1
          SE i ≤ j
            TROCA(L[i],L[j])
            i = i + 1, j = j - 1
        TROCA(L[INICIO], L[j])
        DEVOLVA j
      FIM {PARTIÇÃO}
```

Melhor Caso	Caso Médio	Pior Caso
$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$

2.6 SHELLSORT (GAP)

Esse algoritmo é uma extensão ou refinamento do algoritmo do InsertionSort, contornando o problema que ocorre quando o menor item de um vetor está na posição mais à direita. Ademais, difere desse último pelo fato de, em vez de considerar o vetor a ser ordenado como um único segmento, **ele considera vários segmentos e aplica o InsertionSort em cada um deles.**

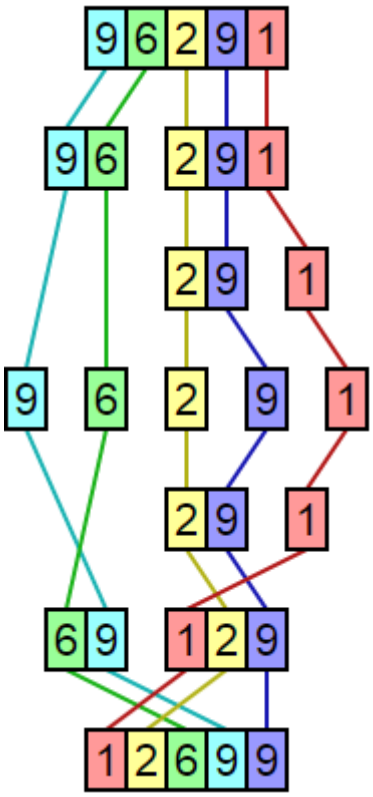
É o algoritmo mais eficiente dentre os de ordem quadrática. Nesse método, as comparações e as trocas são feitas conforme determinada distância *(gap)* entre dois elementos, de modo que, se gap = 6, há comparação entre o 1º e 7º elementos ou entre o 2º e 8º elementos e assim sucessivamente, repetindo até que as últimas comparações e trocas tenham sido efetuadas e o gap tenha chegado a 1.

```
ALGORITMO SHELLSORT
  ENTRADA: UM VETOR L COM N POSIÇÕES
  SAÍDA: O VETOR L EM ORDEM CRESCENTE

  H = 1
  ENQUANTO H < n FAÇA H = 3 * H + 1
  FAÇA
    H = H / 3              // divisão inteira
    PARA i = H até n - 1 // Inserção adaptado para h-listas
      PIVO = L[i]
      j = i - H
      ENQUANTO j ≥ 0 e L[j] > PIVO
        L[j + H] = L[j]
        j= j - H
      L[j + H] = PIVO
  ENQUANTO H > 1
     FIM {SHELLSORT}
```

Melhor Caso	Caso Médio	Pior Caso
$\Omega(n \log n)$	Depende do *gap*	$O(n^2)$

2.7 MERGESORT (MESCLA)



Esse algoritmo é baseado na estratégia de resolução de problemas conhecida como *divisão-e-conquista*. Essa técnica consiste basicamente em decompor a instância a ser resolvida em instâncias menores do mesmo tipo de problema, resolver tais instâncias (em geral, recursivamente) e por fim utilizar as soluções parciais para obter uma solução da instância original.

Naturalmente, nem todo problema pode ser resolvido através de divisão e conquista. Para que seja viável aplicar essa técnica a um problema, ele deve possuir duas propriedades estruturais. **O problema deve ser decomponível**, i.e., deve ser possível decompor qualquer instância não trivial do problema em instâncias menores do mesmo tipo de problema.

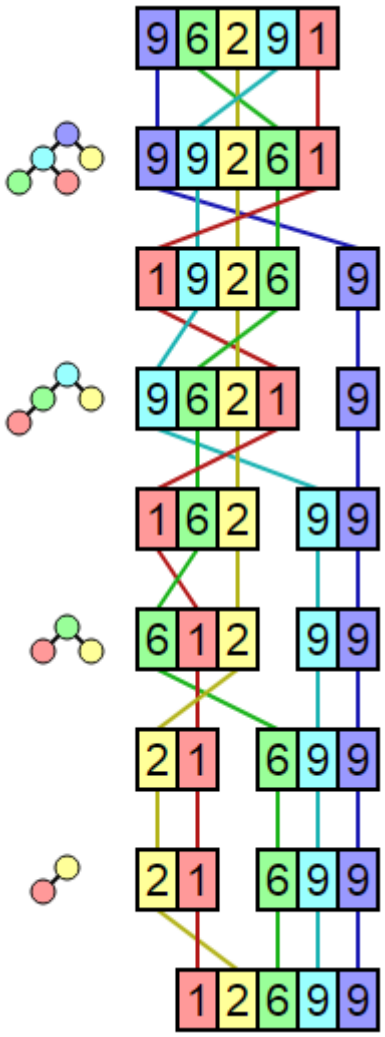
Além disso, deve ser sempre possível utilizar as soluções obtidas com a resolução das instâncias menores para chegar a uma solução da instância original. No MergeSort, divide-se a lista em duas metades. Essas metades são ordenadas recursivamente (usando o próprio MergeSort) e depois são intercaladas. Abaixo segue uma possível solução:

```
ALGORITMO MERGESORT
  ENTRADA: UM VETOR L E AS POSIÇÕES INICIO E FIM
  SAÍDA: O VETOR L EM ORDEM CRESCENTE DA POSIÇÃO INICIO ATÉ
         A POSIÇÃO FIM

  SE inicio < fim
    meio = (inicio + fim)/2          // divisão inteira
    SE inicio < meio
      MERGESORT(L, inicio, meio)
    SE meio + 1 < fim
      MERGESORT(L, meio + 1, fim)
    MERGE(L, inicio, meio, fim)
FIM {MERGESORT}
```

Melhor Caso	Caso Médio	Pior Caso
Θ(n log n)	Θ(n log n)	Θ(n log n)

2.8 HEAPSORT (MAX HEAP)



Esse algoritmo utiliza uma estrutura de dados chamada heap, para ordenar os elementos à medida que os insere na estrutura.

Assim, ao final das inserções, os elementos podem ser sucessivamente removidos da raiz da heap, na ordem desejada.

Essa estrutura pode ser representada como uma árvore ou como um vetor. *Entenderam?* **Inicialmente, insere-se os elementos da lista em um heap.**

Em seguida, fazemos sucessivas remoções do menor elemento do heap, colocando os elementos removidos do heap de volta na lista – a lista estará então em ordem crescente.

O heapsort é um algoritmo de ordenação em que a sua estrutura auxiliar de armazenamento fora do arranjo de entrada é constante durante toda a sua execução.

```
ALGORITMO HEAP SORT
  ENTRADA: UM VETOR L COM N POSIÇÕES
  SAÍDA: O VETOR L EM ORDEM CRESCENTE

inicialize um HBC H com n posições
PARA i = 0 até n - 1
  INSERE_HBC(H, L[i])
PARA i = 0 até n - 1
  L[i] = REMOVE_MENOR(H)
FIM {HEAP SORT}
```

Melhor Caso	Caso Médio	Pior Caso
O(n log n)	O(n log n)	O(n log n)

2.9 RESUMÃO DOS TERMOS CHAVES

- **Bubble Sort**: Compare **um elemento com o seguinte**;
- **Insertion Sort**: Tome da parte desordenada, e **insira na posição certa** da ordenada
- **Selection Sort**: **Selecione o maior** da desordenada e coloque no começo da ordenada (ou **selecione o menor** da desordenada e coloque no final da ordenada)
- **Quick Sort**: Escolha um **pivô** e divida os elementos em 2 conjuntos – maiores e menores que o pivô. Repita para os conjuntos.
- **Shell Sort**: Escolha um **GAP**, compare os elementos separados pelo **GAP**, e vá diminuindo o **GAP**.
- **Merge Sort:** Vá separando os conjuntos na metade. Quando chegar em 2 elementos, **volte fazendo a mescla** de forma ordenada.
- **Heap Sort:** Forme um HEAP, aplique o MAX HEAP e remova a RAIZ.

2.10 RESUMÃO DAS COMPLEXIDADES

ALGORITMO	MELHOR CASO	CASO MÉDIO	PIOR CASO
BubbleSort	$O(n)$	$O(n^2)$	$O(n^2)$
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$
QuickSort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
ShellSort	$O(n \log n)$	Depende do *gap*	$O(n^2)$
MergeSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
HeapSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

2.11 EXERCÍCIOS COMENTADOS: ORDENAÇÃO

1. **(FGV / Analista Censitário (IBGE) / 2017 / Desenvolvimento de Aplicações / Análise de Sistemas)**

O algoritmo de ordenação baseado em vários percursos sobre o *array*, realizando, quando necessárias, trocas entre pares de elementos consecutivos denomina-se método:
a) das trocas (*exchange sort*);
b) da inserção (*insertion sort*);
c) da bolha (*bubble sort*);
d) da seleção (*selection sort*);
e) da permuta (*permutation sort*).

Gabarito: Letra C.

a) das trocas (*exchange sort*);
ERRADO. É um primo do bubble-sort, só muda a forma de comparação. O bubble compara uma posição com a seguinte, e o exchange compara a mesma posição com todas as seguintes.

b) da inserção (*insertion sort*);

ERRADO. O insertion-sort separa o array entre parte ordenada e parte desordenada (inicialmente a parte ordenada é apenas o 1o. elemento, e todos os outros são a parte desordenada). Depois toma item por item da parte desordenada, e posicionando-o corretamente na parte ordenada da coleção.

c) da bolha (*bubble sort*);
CERTO. Compara um item com o posterior, trocando-os se estiverem na ordem errada.

d) da seleção (*selection sort*);
ERRADO. O selection-sort separa o array entre parte ordenada e parte desordenada (inicialmente a parte ordenada está vazia, e todos os elementos estão na parte desordenada). Depois toma o maior item da parte desordenada, e posicionando-o no início na parte ordenada.

e) da permuta (*permutation sort*).
ERRADO. Esse é o método mais "louco" de ordenação... não deve nem ser levado a sério. Consiste em permutar os elementos, e testar para ver se estão em ordem. Se não tiver, faz outra permutação, e testa, e fica repetindo essa tentativa e erro até que a coleção esteja ordenada.

Portanto, a **Letra C** é a resposta correta. 1

2. (FGV / Analista Censitário (IBGE) / 2017 / Desenvolvimento de Aplicações - WEB Mobile / Análise de Sistemas)

Considere o seguinte algoritmo, responsável por realizar a ordenação de um array de dados.

```
public int[] mySortingAlgorithm (int[] date){
  int size = data.length;
  int tmp = 0;
  for(int i = 0;i<size;i++){
   for(int j = (size-1);j>=(i+1);j--){
     if(data[j]<data[j-1]){
        tmp = data[j];
        data[j]=data[j-1];
        data[j-1]=tmp;
      }
    }
  }
  return data;
}
```

Podemos afirmar que o método de ordenação utilizado pelo algoritmo é o:
a) quickSort;
b) insertionSort;
c) mergeSort;
d) shellSort;
e) bubbleSort.

Gabarito: **Letra E**.

Analisando o algoritmo, vemos que ele navega de trás para frente do array, comparando sempre um número com o anterior, e trocando-os de posição caso estejam na ordem errada (vide o trecho abaixo):

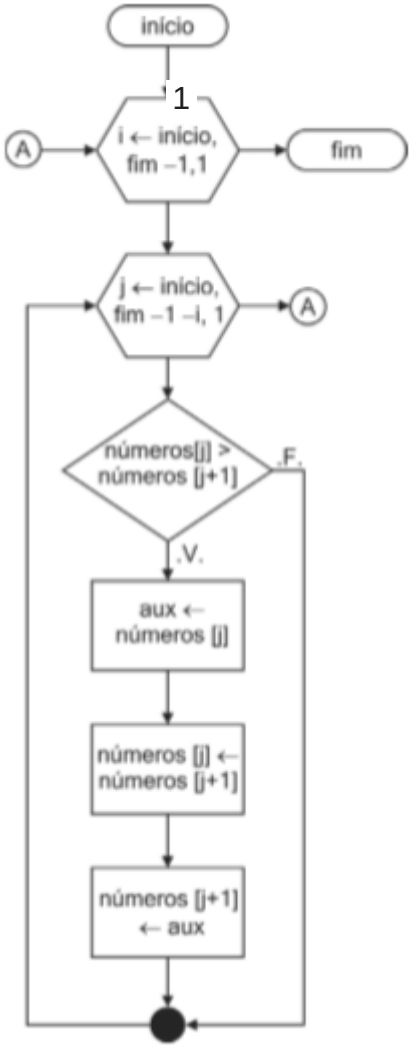
```
if(data[j]<data[j-1]){ //se o item na posição j
                       //for maior que na posição j-1
   tmp = data[j];
```

```
    data[j]=data[j-1];  // troca os itens de lugar
    data[j-1]=tmp;
}
```

Essa é uma variação bem comum do algoritmo Bubble Sort - Compara um item com o seguinte, e troca-os se necessário.

Portanto, a **Letra E** está correta.

3. (CESPE / Técnico Judiciário (TRE TO) / 2017 / Programação de Sistemas / Apoio Especializado)



Se, no fluxograma precedente, *início* indica o primeiro elemento do vetor e *fim*, o último elemento, então, para o vetor [11,6,2,7,8,3,5], o resultado final é

a) [2,7,3,5].
b) [11,7,3,5].
c) [11,6,2,7,8,3,5].
d) [2,3,5,6,7,8,11].
e) [6,2,8,3,5].

Gabarito: **Letra D**.

O algoritmo itera da primeira posição até a penúltima, sempre comparando a posição em evidência com a próxima. Se o elemento atual for maior que o seu sucessor, troca os dois de lugar.

Pessoal, esse é o algoritmo de ordenação conhecido como **Bubble Sort**, ou **Método da Bolha**. Ele ordena uma estrutura de dados, sem remover nenhum elemento.

Portanto, tendo como entrada o vetor [11,6,2,7,8,3,5], o resultado será o mesmo vetor com os elementos ordenados: **[2,3,5,6,7,8,11], Letra D**.

DICA: Sempre que você identificar em um algoritmo a comparação var[j] > var[j+1], desconfie que trata-se do Bubble Sort. Para confirmar, verifique se o bloco que segue esse teste troca os elementos de posição. Esse é o algoritmo mais cobrado em provas!

4. (FCC / Assistente Técnico em Tecnologia da Informação de Defensoria (DPE AM) / 2018 / / Programador)

7

Para responder à questão a seguir, considere a estratégia de ordenação apresentada em Java abaixo.

```
 private static void ordena(int[] vetor, int inicio, int fim) {
     if (inicio < fim) {
             int posicaoP = separar(vetor, inicio, fim);
             ordena(vetor, inicio, posicaoP - 1);
             ordena(vetor, posicaoP + 1, fim);
     }
}

private static int separar(int[] vetor, int inicio, int fim) {
     int P = vetor[inicio];
     int i = inicio + 1, f = fim;
     while (i <= f) {
             if (vetor[i] <= P)
                     i++;
             else if (P < vetor[f])
                     f--;
             else { int troca = vetor[i];
                     vetor[i] = vetor[f];
                     vetor[f] = troca;
                     i++;
                     f--;
             }
     }
     vetor[inicio] = vetor[f];
     vetor[f] = P;
     return f;
}
```

A estratégia apresentada em Java é o método de ordenação

a) Bubblesort.
b) Mergesort.
c) Insertion Sort.

d) Quicksort.
e) Heapsort.

Gabarito: **Letra D**

A estratégia está baseada na escolha de um pivô, conforme acontece na declaração: `int P = vetor[inicio];`

Portanto, podemos eliminar:

a) ~~Bubblesort~~. Compara um número com o seguinte, sem pivô.
b) ~~Mergesort~~. Separa em sub-vetores, ordenando-os. Volta comparando e fazendo uma fusão dos vetores ordenados. Não usa pivô.
e) ~~Heapsort~~. Ordena recursivamente a partir de um Max Heap. Não está sendo utilizado nenhum heap na solução. Também não utiliza pivô.

Sobram a letra C e D.

C

O Insertion Sort normalmente faz a análise somente em um sentido, e o código em análise olha o início e o fim da estrutura, caminhando incrementando o início (i++) e decrementando o fim (f--) para o posicionamento correto do pivot. Esse já é motivo suficiente para eliminar o Insertion Sort, e, portanto, marcar **letra D - Quicksort**.

Para quem quer saber exatamente o nome desse algoritmo, temos aqui um Quicksort com a Partição de Hoare.

5. (FCC / Assistente Técnico em Tecnologia da Informação de Defensoria (DPE AM) / 2018 / / Programador)

Para responder à questão a seguir, considere a estratégia de ordenação apresentada em Java abaixo.

```
 private static void ordena(int[] vetor, int inicio, int fim) {
      if (inicio < fim) {
               int posicaoP = separar(vetor, inicio, fim);
               ordena(vetor, inicio, posicaoP - 1);
               ordena(vetor, posicaoP + 1, fim);
      }
}

private static int separar(int[] vetor, int inicio, int fim) {
      int P = vetor[inicio];
      int i = inicio + 1, f = fim;
      while (i <= f) {
               if (vetor[i] <= P)
                        i++;
               else if (P < vetor[f])
                        f--;
               else { int troca = vetor[i];
                        vetor[i] = vetor[f];
                        vetor[f] = troca;
                        i++;
                        f--;
               }
```

```
        }
        vetor[inicio] = vetor[f];
        vetor[f] = P;
        return f;
}
```

Considerando que N é número de elementos do vetor a ser ordenado, a estratégia de ordenação apresentada em Java

a) também é conhecida como método de ordenação por intercalação e possui uma versão para unir dois vetores já ordenados.

b) tem complexidade $O(N^2)$ no pior caso e no caso médio, mas apresenta complexidade O(N) no melhor caso.

c) faz um número fixo de comparações dado por $\log_2 N$, independente dos valores do vetor original. Isso é garantido pelas chamadas recursivas ao método ordena().

d) utiliza o método separar() para dividir o vetor original em 2 sublistas de igual tamanho. Isso garante que mesmo no pior caso o método realize Nlog2N comparações.

e) utiliza o método separar() para fazer a partição do vetor, por meio da seleção de um elemento chamado pivô. A escolha do pivô é crucial para o bom desempenho do método, já que a fase de partição é a parte crítica do algoritmo.

Gabarito: **Letra E**.

a) também é conhecida como método de ~~ordenação por intercalação~~ e ~~possui uma versão para unir dois vetores já ordenados~~.
ERRADO. A definição de ordenação por intercalação (merge sort) está correta, mas o método utilizado é o **Quicksort**.

b) tem complexidade $O(N^2)$ no pior caso e no caso médio, mas apresenta complexidade ~~O(N)~~ no melhor caso.
ERRADO. Para o **Quicksort**, o pior caso está correto $O(N^2)$, mas o melhor caso e o caso médio possuem complexidade O(N log N).

c) faz um ~~número fixo de comparações dado por $\log_2 N$~~, independente dos valores do vetor original. Isso é garantido pelas chamadas recursivas ao método ordena().
ERRADO. O número de comparações do Quicksort será sempre maior que $\log_2 N$.

d) utiliza o método separar() para dividir o vetor original em 2 sublistas ~~de igual tamanho~~. Isso garante que mesmo no pior caso o método realize Nlog2N comparações.
ERRADO. A divisão em 2 arrays não é igualitária, e vai depender da posição correta do pivô no array ordenado. Se a posição correta do pivô for no início, por exemplo, o sub-array da esquerda nem vai existir.

e) utiliza o método separar() para fazer a partição do vetor, por meio da seleção de um elemento chamado pivô. A escolha do pivô é crucial para o bom desempenho do método, já que a fase de partição é a parte crítica do algoritmo.
CORRETO. No Quicksort a escolha do pivô pode ser um algoritmo por si só. Existem diversas possibilidades de partição como, por exemplo, a Hoares e Lomuto. A escolha de pivô e do algoritmo de partição podem mudar completamente a performance do método.

6. (CESGRANRIO / Analista (PETROBRAS) / 2018 / / Sistema Júnior)

Dada a sequência numérica (15,11,16,18,23,5,10,22,21,12) para ordenar pelo algoritmo Selection Sort, qual é a sequência parcialmente ordenada depois de completada a quinta passagem do algoritmo?

a) [15, 11, 16, 18, 12, 5, 10, 21, 22, 23]
b) [15, 11, 5, 10, 12, 16, 18, 21, 22, 23]
c) [15, 11, 16, 10, 12, 5, 18, 21, 22, 23]
d) [10, 11, 5, 12, 15, 16, 18, 21, 22, 23]
e) [12, 11, 5, 10, 15, 16, 18, 21, 22, 23]

Gabarito: **Letra B**

O método **selection sort** separa o array em **parte ordenada** (normalmente à direita) e **parte desordenada** (normalmente à esquerda). O método consiste em pegar o maior número da parte desordenada e colocar no início parte ordenada. Pode-se começar a partir do penúltimo valor.

Sendo assim:

Inicial: [15,11,16,18,23,5,10,22,21][12] b
1a. passagem: [15,11,16,18,~~**23**~~,5,10,22,21][12,**23**]
2a. passagem: [15,11,16,18,5,10,~~**22**~~,21][12,**22**,23]
3a. passagem: [15,11,16,18,5,10,~~**21**~~][12,**21**,22,23]
4a. passagem: [15,11,16,~~**18**~~,5,10][12,**18**,21,22,23]
5a. passagem: [15,11,~~**16**~~,5,10][12,**16**,18,21,22,23]

Portanto, na 5a. passagem o vetor estará com a sequencia [15,11,5,10,12,16,18,21,22,23], **Letra B**.

7. (CESPE / Oficial Técnico de Inteligência / 2018 / / Área 9)

O método de ordenação conhecido como quick sort utiliza o maior elemento, o qual é sempre colocado ao final do vetor, para garantir que a ordenação seja realizada em ordem decrescente.

Gabarito: **ERRADO**.

O método *quick sort* utiliza um pivô. Passa pelo array, organizando elementos menores que o pivô à esquerda, e maiores que o pivô à direita. No final de uma iteração, o pivô está na posição correta. Ele repete então o algoritmo na coleção à esquerda e à direita.

O método que passa selecionando o maior elemento e colocando-o em ordem é o ***Selection Sort***. Ess algoritmo separa o vetor em parte desordenada (à esquerda) e parte ordenada (à direita). A cada iteração, o posicionamento do maior elemento da parte desordenada ocorre no início da parte ordenada (ou final da parte desordenada, que dá no mesmo).

Reescrevendo o item:

O método de ordenação conhecido como ~~*quick sort*~~ ***selection sort*** utiliza o maior elemento **do sub-vetor desordenado**, o qual é sempre colocado ~~ao final do vetor~~ **no início**

do sub-vetor ordenado para garantir que a ordenação seja realizada em ordem decrescente.

8. (AOCP / Técnico em Gestão de Infraestrutura (SUSIPE) / 2018 / / Gestão de Informática)

Assinale a alternativa correta a respeito dos principais algoritmos de ordenação.
a) O algoritmo de ordenação QuickSort é um algoritmo eficiente, porém deve ser implementado visando a uma boa escolha do elemento pivô.
b) O algoritmo de ordenação por inserção tem uma implementação cara, porém com um desempenho estável.
c) O algoritmo de ordenação HeapSort é um algoritmo eficiente em relação ao tempo, porém ineficiente em relação à memória.
d) O algoritmo de ordenação ShellSort tem sua principal desvantagem quando os dados a serem ordenados estão parcialmente ordenados.
e) O algoritmo de ordenação por seleção tem uma implementação cara, porém com desempenho estável.

Gabarito: **Letra A**.

a) O algoritmo de ordenação QuickSort é um algoritmo eficiente, porém deve ser implementado visando a uma boa escolha do elemento pivô.
CORRETO. Essa é uma característica marcante do método QuickSort. Ele trabalha ordenando a partir da escolha de um pivô, colocando os itens menores à esquerda, e maiores à direita. Dessa forma, a escolha do pivô tem um papel central no método.

b) O algoritmo de ordenação por inserção tem uma ~~implementação cara~~, porém com um desempenho estável.
INCORRETO. O *insertion sort* têm implementação muito simples: toma o próximo valor desordenado, e insere-o em posição na coleção ordenada.

c) O algoritmo de ordenação HeapSort é um algoritmo eficiente em relação ao tempo, porém ineficiente em relação à memória.
INCORRETO. O *heap sort* é bem eficiente com relação à memória, uma vez que a estrutura de *Heap* utilizada pelo método pode ser feita sobre o próprio vetor de entrada, sem a necessidade de alocação extra de memória.

d) O algoritmo de ordenação ShellSort tem sua principal desvantagem quando os dados a serem ordenados estão parcialmente ordenados.
INCORRETO. O *shell sort* faz a ordenação a partir da comparação de 2 elementos no array de distâncias (GAPs) variáveis. O pior caso depende do posicionamento dos elementos com relação aos GAPs. O método que tem seu pior caso em arrays ordenados (ou quase ordenados) é o *quick sort*.

e) O algoritmo de ordenação por seleção tem uma implementação cara, porém com desempenho estável.

INCORRETO. O *selection sort* têm implementação muito simples: toma o maior valor desordenado, e insere-o no início da coleção ordenada.

9. (CESGRANRIO / Analista de Sistemas Júnior (TRANSPETRO) / 2018 / / Processos de Negócio)

Analise o algoritmo de ordenação que se segue.

```
def ordenar(dado):
    for passnum in range(len(dado)-1,0,-1):
      for i in range(passnum):
          if dado[i]>dado[i+1]:
             temp = dado[i]
             dado[i] = dado[i+1]
             dado[i+1] = temp
dado = [16,18,15,37,13]
ordenar(dado)
print(dado)
```

Com o uso desse algoritmo, qual é a quantidade de trocas realizadas para ordenar a sequência **dado**?

a) 4
b) 5
c) 6
d) 7
e) 8

Gabarito: **Letra C**.

Analisando o algoritmo, temos que ele percorre a coleção, comparando o de trás para frente e, se o item anterior for maior que o atual, troca os dois de posição. É uma variação do Bubble Sort, analisando de trás para frente. As trocas são feitas entre elementos consecutivos, até que a coleção esteja ordenada. Portanto:

INICIAL: [16,18,15,37,13]
1: TROCA 37,13 - [16,18,15,13,37]
2: TROCA 15,13 - [16,18,13,15,37]
3: TROCA 18,13 - [16,13,18,15,37]
4: TROCA 13,16 - [13,16,18,15,37]
5: TROCA 18,15 - [13,16,15,18,37]
6: TROCA 16,15 - **[13,15,16,18,37]**

Pronto! A coleção está organizada depois de 6 trocas. Portanto, **Letra C**.

Curiosidade: A linguagem utilizada na questão é Python.

10. (CESGRANRIO / Escriturário (BB) / 2018)

O programa a seguir, em Python, implementa o algoritmo do método de bolha, imprimindo o resultado de cada passo.

```
def bolha(lista):
    for passo in range(len(lista)-1,0,-1):
        for i in range(passo):
            if lista[i]>lista[i+1]:
                lista[i],lista[i+1]=lista[i+1],lista[i]
        print(lista)
```

Qual será a quarta linha impressa para a chamada **bolha([4, 3, 1, 9, 8, 7, 2, 5])** ?

a) [3, 1, 4, 8, 7, 2, 5, 9]
b) [1, 3, 4, 7, 2, 5, 8, 9]
c) [1, 2, 3, 4, 5, 7, 8, 9]
d) [1, 3, 2, 4, 5, 7, 8, 9]
e) [1, 3, 4, 2, 5, 7, 8, 9]

Gabarito: **Letra D.**

Analisando iteração por iteração da função bolha (a cada iteração, mais uma posição ao final do array está correta, e não precisa mais ser testada):

```
# Iteração 1
  -  Começa com:  [4, 3, 1, 9, 8, 7, 2, 5]
   [4, 3, 1, 9, 8, 7, 2, 5]  - Troca  4  <->  3
   [3, 4, 1, 9, 8, 7, 2, 5]  - Troca  4  <->  1
   [3, 1, 4, 9, 8, 7, 2, 5]  - Mantem 4   -   9
   [3, 1, 4, 9, 8, 7, 2, 5]  - Troca  9  <->  8
   [3, 1, 4, 8, 9, 7, 2, 5]  - Troca  9  <->  7
   [3, 1, 4, 8, 7, 9, 2, 5]  - Troca  9  <->  2
   [3, 1, 4, 8, 7, 2, 9, 5]  - Troca  9  <->  5
  - Termina com:  [3, 1, 4, 8, 7, 2, 5, 9]
# Iteração 2
  -  Começa com:  [3, 1, 4, 8, 7, 2, 5, 9]
   [3, 1, 4, 8, 7, 2, 5, 9]  - Troca  3  <->  1
   [1, 3, 4, 8, 7, 2, 5, 9]  - Mantem 3   -   4
   [1, 3, 4, 8, 7, 2, 5, 9]  - Mantem 4   -   8
   [1, 3, 4, 8, 7, 2, 5, 9]  - Troca  8  <->  7
   [1, 3, 4, 7, 8, 2, 5, 9]  - Troca  8  <->  2
   [1, 3, 4, 7, 2, 8, 5, 9]  - Troca  8  <->  5
  - Termina com:  [1, 3, 4, 7, 2, 5, 8, 9]
# Iteração 3
  -  Começa com:  [1, 3, 4, 7, 2, 5, 8, 9]
   [1, 3, 4, 7, 2, 5, 8, 9]  - Mantem 1   -   3
   [1, 3, 4, 7, 2, 5, 8, 9]  - Mantem 3   -   4
   [1, 3, 4, 7, 2, 5, 8, 9]  - Mantem 4   -   7
   [1, 3, 4, 7, 2, 5, 8, 9]  - Troca  7  <->  2
   [1, 3, 4, 2, 7, 5, 8, 9]  - Troca  7  <->  5
  - Termina com:  [1, 3, 4, 2, 5, 7, 8, 9]
# Iteração 4
  -  Começa com:  [1, 3, 4, 2, 5, 7, 8, 9]
   [1, 3, 4, 2, 5, 7, 8, 9]  - Mantem 1   -   3
   [1, 3, 4, 2, 5, 7, 8, 9]  - Mantem 3   -   4
   [1, 3, 4, 2, 5, 7, 8, 9]  - Troca  4  <->  2
   [1, 3, 2, 4, 5, 7, 8, 9]  - Mantem 4   -   5
  - Termina com:  [1, 3, 2, 4, 5, 7, 8, 9] - Letra D.
```

11. (FCC / Analista Judiciário (TRF 5ª Região) / 2017 / Informática - Desenvolvimento / Apoio Especializado)

O algoritmo QuickSort usa uma técnica conhecida por divisão e conquista, onde problemas complexos são reduzidos em problemas menores para se tentar chegar a uma solução.

A complexidade média deste algoritmo em sua implementação padrão e a complexidade de pior caso são, respectivamente,

a) $O(n-1)$ e $O(n^3)$.
b) $O(n^2)$ e $O(n \log n^2)$.
c) $O(n^2)$ e $O(n^3)$.
d) $O(n)$ e $O(n^2)$.
e) $O(n \log n)$ e $O(n^2)$.

Gabarito: **LETRA E.**

ALGORITMO	MELHOR CASO	CASO MÉDIO	PIOR CASO
BubbleSort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
InsertionSort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
SelectionSort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
QuickSort	**$\Omega(n \log n)$**	**$\Theta(n \log n)$**	**$O(n^2)$**
ShellSort	$\Omega(n \log n)$	Depende do *gap*	$O(n^2)$
MergeSort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$
HeapSort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$

12. (FCC / Assistente Técnico em Tecnologia da Informação de Defensoria (DPE AM) / 2018 / / Programador)

Para ordenar um vetor com N elementos, o método de ordenação Seleção (Selection Sort) faz o seguinte número de comparações:
a) $(N^2 - N)/2$, sendo muito lento e inadequado para valores grandes de N.
b) $\log_2(N^2 + N)$ no melhor caso.
c) $(N^2 + N - 1)/2$ no caso médio, ficando lento para valores grandes de N.
d) $(N - 1)$ quando o vetor já está originalmente ordenado.
e) $(N^2 + N)/4$ no pior caso, sendo melhor que o pior caso do Bolha (Bubble Sort) pois faz menos trocas.

Gabarito: **Letra A**

O Selection Sort percorre a parte desordenada do vetor selecionando o maior elemento (daí o nome *selection*), e posicionando no início da parte ordenada. Ele precisa fazer um número de comparações que depende do tamanho da parte desordenada do vetor.

Na primeira passagem, precisa fazer N-1 comparações. Na segunda, N-2. Na terceira, N-3, e assim sucessivamente até só fazer 1 comparação. Portanto, temos:

Comparações = (N-1) + (N-2) + ... + 3 + 2 + 1

É uma progressão aritmética de (N-1) elementos, portanto, a soma será o número de elementos (N-1), vezes média do primeiro elemento (N-1) com o último elemento (1):

Comparações = (N-1)*((N-1) + (1))/2 = (N-1)*(N)/2 = (N^2 − N)/2.

É uma complexidade quadrática e, portanto, inadequada para valores grandes de N.

13. (CESGRANRIO - 2010 – BACEN – Analista de Sistemas)

Uma fábrica de software foi contratada para desenvolver um produto de análise de riscos. Em determinada funcionalidade desse software, é necessário realizar a ordenação de um conjunto formado por muitos números inteiros. Que algoritmo de ordenação oferece melhor complexidade de tempo (Big O notation) no pior caso?

a) Merge sort
b) Insertion sort
c) Bubble sort
d) Quick sort
e) Selection sort

Gabarito: **Letra A**

Algoritmo	Melhor Caso	Caso Médio	Pior Caso
BubbleSort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
InsertionSort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
SelectionSort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
QuickSort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$
ShellSort	$\Omega(n \log n)$	Depende do *gap*	$O(n^2)$
MergeSort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$
HeapSort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$

14. (FGV - 2013 – MPE/MS – Analista de Sistemas)

Assinale a alternativa que indica o algoritmo de ordenação capaz de funcionar em tempo O(n) para alguns conjuntos de entrada.

a) Selectionsort (seleção)
b) Insertionsort (inserção)
c) Merge sort
d) Quicksort
e) Heapsort

Gabarito: **Letra B**.

Algoritmo	Melhor Caso	Caso Médio	Pior Caso
BubbleSort	$O(n)$	$O(n^2)$	$O(n^2)$
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$
QuickSort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
ShellSort	$O(n \log n)$	Depende do *gap*	$O(n^2)$
MergeSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
HeapSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

15. (CESGRANRIO - 2011 - PETROBRÁS – Analista de Sistemas – III)

O tempo médio do QuickSort é de ordem igual ao tempo médio do MergeSort.

Gabarito: **CERTO**.

Algoritmo	Melhor Caso	Caso Médio	Pior Caso
BubbleSort	$O(n)$	$O(n^2)$	$O(n^2)$
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$
QuickSort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
ShellSort	$O(n \log n)$	Depende do *gap*	$O(n^2)$
MergeSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
HeapSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

3 LISTA DE EXERCÍCIOS

3.1 LISTA DE EXERCÍCIOS: COMPLEXIDADE DE ALGORITMOS

1. **(FGV / Analista Censitário (IBGE) / 2017 / Desenvolvimento de Aplicações - WEB Mobile / Análise de Sistemas)**

Para projetar algoritmos eficientes um desenvolvedor deve estar preocupado com a complexidade deste algoritmo, desde sua concepção.

Considere a seguinte função T(n) que mede os recursos (ex. tempo de execução) que um algoritmo necessita no pior caso para processar uma entrada qualquer de tamanho n:

T(n) = O(log(n))

Sabendo que O(log(n)) é a ordem da complexidade de tempo do algoritmo seguindo a notação "big O", é correto afirmar que este algoritmo tem complexidade de ordem:
a) constante;
b) sublinear;
c) linear;
d) polinomial;
e) exponencial.

2. **(FCC / Analista Judiciário (TRF 5ª Região) / 2017 / Informática - Desenvolvimento / Apoio Especializado)**

Considere o algoritmo abaixo.

```
static int fibonacci(int n) {
  if (n <= 1) {
     return n;
  }
  return fibonacci(n - 2) + fibonacci(n - 1);
}
```

A complexidade deste algoritmo, na notação Big O, é
a) $O(2^n)$.
b) $O(n^2)$.
c) O(n).
d) O(log(n)).
e) $O(n^4)$.

3. (FGV / Analista Legislativo (ALERO) / 2018 / Infraestrutura de Redes e Comunicação / Tecnologia da Informação)

Considere a Sequência de Fibonacci (0, 1, 1, 2, 3, 5, 8, 13, ...), onde os dois primeiros termos valem 0 e 1 respectivamente, e cada termo seguinte é a soma de seus dois predecessores.

O pseudocódigo a seguir apresenta um algoritmo simples para o cálculo do N-ésimo termo dessa sequência.

```
function fibo (N)
if n = 1 then
     return 0
elif n = 2 then
     return 1
else
       penultimo := 0
       ultimo := 1
       for i := 3 until N do
            atual := penultimo + ultimo
            penultimo := ultimo
            ultimo := atual
       end for
       return atual
end if
```

Assinale a opção que mostra a complexidade desse algoritmo.
a) O(n/2)
b) O(n)
c) O(n2)
d) O(log n)
e) O(2n)

3.2 LISTA DE EXERCÍCIOS: ORDENAÇÃO

4. (FGV / Analista Censitário (IBGE) / 2017 / Desenvolvimento de Aplicações / Análise de Sistemas)

O algoritmo de ordenação baseado em vários percursos sobre o *array*, realizando, quando necessárias, trocas entre pares de elementos consecutivos denomina-se método:
a) das trocas (*exchange sort*);
b) da inserção (*insertion sort*);
c) da bolha (*bubble sort*);
d) da seleção (*selection sort*);
e) da permuta (*permutation sort*).

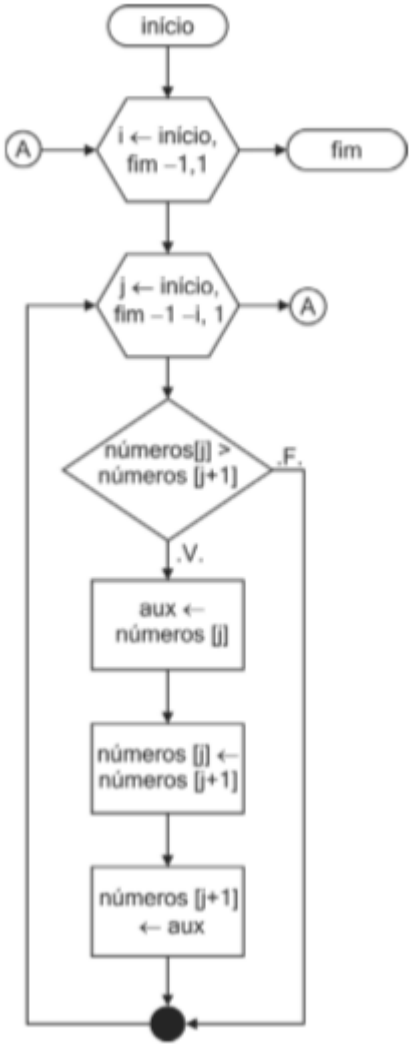
5. (FGV / Analista Censitário (IBGE) / 2017 / Desenvolvimento de Aplicações - WEB Mobile / Análise de Sistemas)

Considere o seguinte algoritmo, responsável por realizar a ordenação de um array de dados.

```
public int[] mySortingAlgorithm (int[] date){
  int size = data.length;
  int tmp = 0;
  for(int i = 0;i<size;i++){
   for(int j = (size-1);j>=(i+1);j--){
     if(data[j]<data[j-1]){
        tmp = data[j];
        data[j]=data[j-1];
        data[j-1]=tmp;
      }
    }
  }
  return data;
}
```

Podemos afirmar que o método de ordenação utilizado pelo algoritmo é o:
a) quickSort;
b) insertionSort;
c) mergeSort;
d) shellSort;
e) bubbleSort.

6. (CESPE / Técnico Judiciário (TRE TO) / 2017 / Programação de Sistemas / Apoio Especializado)



Se, no fluxograma precedente, *início* indica o primeiro elemento do vetor e *fim*, o último elemento, então, para o vetor [11,6,2,7,8,3,5], o resultado final é
a) [2,7,3,5].
b) [11,7,3,5].
c) [11,6,2,7,8,3,5].
d) [2,3,5,6,7,8,11].
e) [6,2,8,3,5].

7. (FCC / Assistente Técnico em Tecnologia da Informação de Defensoria (DPE AM) / 2018 / / Programador)

Para responder à questão a seguir, considere a estratégia de ordenação apresentada em Java abaixo.

```
 private static void ordena(int[] vetor, int inicio, int fim) {
     if (inicio < fim) {
             int posicaoP = separar(vetor, inicio, fim);
             ordena(vetor, inicio, posicaoP - 1);
             ordena(vetor, posicaoP + 1, fim);
     }
}
```

```
private static int separar(int[] vetor, int inicio, int fim) {
      int P = vetor[inicio];
      int i = inicio + 1, f = fim;
      while (i <= f) {
              if (vetor[i] <= P)
                      i++;
              else if (P < vetor[f])
                      f--;
              else { int troca = vetor[i];
                      vetor[i] = vetor[f];
                      vetor[f] = troca;
                      i++;
                      f--;
              }
      }
      vetor[inicio] = vetor[f];
      vetor[f] = P;
      return f;
}
```

A estratégia apresentada em Java é o método de ordenação
a) Bubblesort.
b) Mergesort.
c) Insertion Sort.
d) Quicksort.
e) Heapsort.

8. (FCC / Assistente Técnico em Tecnologia da Informação de Defensoria (DPE AM) / 2018 / / Programador)

Para responder à questão a seguir, considere a estratégia de ordenação apresentada em Java abaixo.

```
 private static void ordena(int[] vetor, int inicio, int fim) {
      if (inicio < fim) {
              int posicaoP = separar(vetor, inicio, fim);
              ordena(vetor, inicio, posicaoP - 1);
              ordena(vetor, posicaoP + 1, fim);
      }
}

private static int separar(int[] vetor, int inicio, int fim) {
      int P = vetor[inicio];
      int i = inicio + 1, f = fim;
      while (i <= f) {
              if (vetor[i] <= P)
                      i++;
              else if (P < vetor[f])
                      f--;
              else { int troca = vetor[i];
                      vetor[i] = vetor[f];
                      vetor[f] = troca;
```

```
                        i++;
                        f--;
                }
        }
        vetor[inicio] = vetor[f];
        vetor[f] = P;
        return f;
}
```

Considerando que N é número de elementos do vetor a ser ordenado, a estratégia de ordenação apresentada em Java

a) também é conhecida como método de ordenação por intercalação e possui uma versão para unir dois vetores já ordenados.

b) tem complexidade $O(N^2)$ no pior caso e no caso médio, mas apresenta complexidade O(N) no melhor caso.

c) faz um número fixo de comparações dado por $\log_2 N$, independente dos valores do vetor original. Isso é garantido pelas chamadas recursivas ao método ordena().

d) utiliza o método separar() para dividir o vetor original em 2 sublistas de igual tamanho. Isso garante que mesmo no pior caso o método realize Nlog2N comparações.

e) utiliza o método separar() para fazer a partição do vetor, por meio da seleção de um elemento chamado pivô. A escolha do pivô é crucial para o bom desempenho do método, já que a fase de partição é a parte crítica do algoritmo.

9. (CESGRANRIO / Analista (PETROBRAS) / 2018 / / Sistema Júnior)

Dada a sequência numérica (15,11,16,18,23,5,10,22,21,12) para ordenar pelo algoritmo Selection Sort, qual é a sequência parcialmente ordenada depois de completada a quinta passagem do algoritmo?

a) [15, 11, 16, 18, 12, 5, 10, 21, 22, 23]
b) [15, 11, 5, 10, 12, 16, 18, 21, 22, 23]
c) [15, 11, 16, 10, 12, 5, 18, 21, 22, 23]
d) [10, 11, 5, 12, 15, 16, 18, 21, 22, 23]
e) [12, 11, 5, 10, 15, 16, 18, 21, 22, 23]

10. (CESPE / Oficial Técnico de Inteligência / 2018 / / Área 9)

O método de ordenação conhecido como quick sort utiliza o maior elemento, o qual é sempre colocado ao final do vetor, para garantir que a ordenação seja realizada em ordem decrescente.

11. (AOCP / Técnico em Gestão de Infraestrutura (SUSIPE) / 2018 / / Gestão de Informática)

Assinale a alternativa correta a respeito dos principais algoritmos de ordenação.

a) O algoritmo de ordenação QuickSort é um algoritmo eficiente, porém deve ser implementado visando a uma boa escolha do elemento pivô.

b) O algoritmo de ordenação por inserção tem uma implementação cara, porém com um desempenho estável.

c) O algoritmo de ordenação HeapSort é um algoritmo eficiente em relação ao tempo, porém ineficiente em relação à memória.
d) O algoritmo de ordenação ShellSort tem sua principal desvantagem quando os dados a serem ordenados estão parcialmente ordenados.
e) O algoritmo de ordenação por seleção tem uma implementação cara, porém com desempenho estável.

12. (CESGRANRIO / Analista de Sistemas Júnior (TRANSPETRO) / 2018 / / Processos de Negócio)

Analise o algoritmo de ordenação que se segue.

```
def ordenar(dado):
    for passnum in range(len(dado)-1,0,-1):
      for i in range(passnum):
          if dado[i]>dado[i+1]:
             temp = dado[i]
             dado[i] = dado[i+1]
             dado[i+1] = temp
dado = [16,18,15,37,13]
ordenar(dado)
print(dado)
```

Com o uso desse algoritmo, qual é a quantidade de trocas realizadas para ordenar a sequência **dado**?
a) 4
b) 5
c) 6
d) 7
e) 8

13. (CESGRANRIO / Escriturário (BB) / 2018)

O programa a seguir, em Python, implementa o algoritmo do método de bolha, imprimindo o resultado de cada passo.

```
def bolha(lista):
    for passo in range(len(lista)-1,0,-1):
         for i in range(passo):
               if lista[i]>lista[i+1]:
                     lista[i],lista[i+1]=lista[i+1],lista[i]
         print(lista)
```

Qual será a quarta linha impressa para a chamada **bolha([4, 3, 1, 9, 8, 7, 2, 5])** ?
a) [3, 1, 4, 8, 7, 2, 5, 9]
b) [1, 3, 4, 7, 2, 5, 8, 9]
c) [1, 2, 3, 4, 5, 7, 8, 9]
d) [1, 3, 2, 4, 5, 7, 8, 9]
e) [1, 3, 4, 2, 5, 7, 8, 9]

14. (FCC / Analista Judiciário (TRF 5ª Região) / 2017 / Informática - Desenvolvimento / Apoio Especializado)

O algoritmo QuickSort usa uma técnica conhecida por divisão e conquista, onde problemas complexos são reduzidos em problemas menores para se tentar chegar a uma solução.

A complexidade média deste algoritmo em sua implementação padrão e a complexidade de pior caso são, respectivamente,

a) $O(n-1)$ e $O(n^3)$.
b) $O(n^2)$ e $O(n \log n^2)$.
c) $O(n^2)$ e $O(n^3)$.
d) $O(n)$ e $O(n^2)$.
e) $O(n \log n)$ e $O(n^2)$.

15. (FCC / Assistente Técnico em Tecnologia da Informação de Defensoria (DPE AM) / 2018 / / Programador)

Para ordenar um vetor com N elementos, o método de ordenação Seleção (Selection Sort) faz o seguinte número de comparações:
a) $(N^2 - N)/2$, sendo muito lento e inadequado para valores grandes de N.
b) $\log_2(N^2 + N)$ no melhor caso.
c) $(N^2 + N - 1)/2$ no caso médio, ficando lento para valores grandes de N.
d) $(N - 1)$ quando o vetor já está originalmente ordenado.
e) $(N^2 + N)/4$ no pior caso, sendo melhor que o pior caso do Bolha (Bubble Sort) pois faz menos trocas.

16. (CESGRANRIO - 2010 – BACEN – Analista de Sistemas)

Uma fábrica de software foi contratada para desenvolver um produto de análise de riscos. Em determinada funcionalidade desse software, é necessário realizar a ordenação de um conjunto formado por muitos números inteiros. Que algoritmo de ordenação oferece melhor complexidade de tempo (Big O notation) no pior caso?

a) Merge sort
b) Insertion sort
c) Bubble sort
d) Quick sort
e) Selection sort

17. (FGV - 2013 – MPE/MS – Analista de Sistemas)

Assinale a alternativa que indica o algoritmo de ordenação capaz de funcionar em tempo O(n) para alguns conjuntos de entrada.

a) Selectionsort (seleção)
b) Insertionsort (inserção)
c) Merge sort

d) Quicksort
e) Heapsort

18. (CESGRANRIO - 2011 - PETROBRÁS – Analista de Sistemas – III)

O tempo médio do QuickSort é de ordem igual ao tempo médio do MergeSort.

4 GABARITOS

4.1 GABARITO: COMPLEXIDADE DE ALGORITMOS

1. B
2. A
3. B

4.2 GABARITO: ORDENAÇÃO

1. C
2. E
3. D
4. D
5. E
6. B
7. ERRADO
8. A
9. C
10. D
11. E
12. A
13. A
14. B
15. CERTO

40

ESSA LEI TODO MUNDO CONHECE:
PIRATARIA É CRIME.

Mas é sempre bom revisar o porquê e como você pode ser prejudicado com essa prática.

1. 

Professor investe seu tempo para elaborar os cursos e o site os coloca à venda.

2. 

Pirata divulga ilicitamente (grupos de rateio), utilizando-se do anonimato, nomes falsos ou laranjas (geralmente o pirata se anuncia como formador de "grupos solidários" de rateio que não visam lucro).

3. 

Pirata cria alunos fake praticando falsidade ideológica, comprando cursos do site em nome de pessoas aleatórias (usando nome, CPF, endereço e telefone de terceiros sem autorização).

4. 

Pirata compra, muitas vezes, clonando cartões de crédito (por vezes o sistema anti-fraude não consegue identificar o golpe a tempo).

5. 

Pirata fere os Termos de Uso, adultera as aulas e retira a identificação dos arquivos PDF (justamente porque a atividade é ilegal e ele não quer que seus fakes sejam identificados).

6. 

Pirata revende as aulas protegidas por direitos autorais, praticando concorrência desleal e em flagrante desrespeito à Lei de Direitos Autorais (Lei 9.610/98).

7. 

Concurseiro(a) desinformado participa de rateio, achando que nada disso está acontecendo e esperando se tornar servidor público para exigir o cumprimento das leis.

8. 

O professor que elaborou o curso não ganha nada, o site não recebe nada, e a pessoa que praticou todos os ilícitos anteriores (pirata) fica com o lucro.

Deixando de lado esse mar de sujeira, aproveitamos para agradecer a todos que adquirem os cursos honestamente e permitem que o site continue existindo.